All Together Now

Writing Multiuser Application Sofware for ACCESS.bus

With ACCESS.bus, up to 125 users can simultaneously interact on one host computer. A close look at hardware and software architectures prepares you for the ultimate challenge: create your own game!

FEATURE ARTICLE

David Rodgers & Peretz Tzarnotzky

ACCESS.bus is a serial communication protocol between a computer host and its peripheral devices. It provides a simple, uniform, and inexpensive way to connect peripheral devices such as keyboards, mice, joysticks, modems, monitors, and printers to a single computer port.

ACCESS.bus (a bus for connecting ACCESSory devices to a host system) was defined and developed by Digital Equipment Corporation. DEC offered it to the computer industry as an open standard, enabling any vendor to implement it on host systems or in peripheral devices without fees or royalties.

Although ACCESS.bus protocol architecture is an open industry standard, it is controlled and maintained by the ACCESS.bus Industry Group (ABIG). Significant corporate and individual members, representing all facets of the computer industry, participate in the evolution, definition, and proliferation of ACCESS.bus technology.

ACCESS.bus is a system for connecting up to 125 I/O devices to a single port on a host computer. Interface adapters to ISA-based PC systems and SPARCstation SBus systems facilitate easy connection to common hardware installations. ACCESS.bus peripheral I/O devices (keyboards, mice, trackballs, etc.) are already available from leading manufacturers such as Logitech, Key Tronic, and Lexmark, to name a few. Full peripheral and software application development tools are also available for easy ACCESS.bus product development. System-board-level adaptations of ACCESS.bus are currently being evaluated by several personal computer systems manufacturers.

As well, the ACCESS.bus protocol is general enough to accommodate a wide range of specialized vertical applications including point of sale, education, embedded control, virtual reality, and PC-based games. See Figure 1 for the typical ACCESS.bus system.

ACCESS.bus data-transmission technology (clock, data, power, and ground) provides microcontroller integration on various open and proprietary operating platforms and peripherals. From medical imaging and training systems to multiplayer, multimedia computer games, from point-of-sale systems integration and data collection to set-top cable converters, ACCESS.bus has evolved into a versatile and powerful tool.

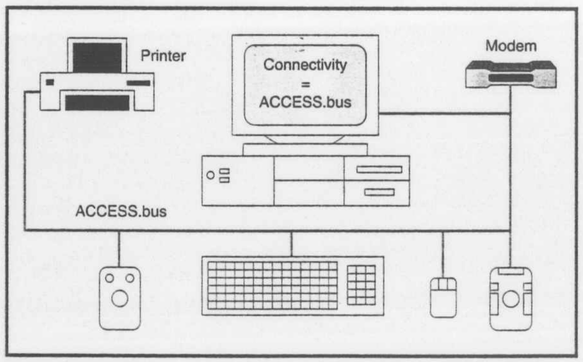

Figure 1—In a typical ACCESS.bus desktop configuration, the usual nest of cables required to connect external peripherals is replaced by a single four-conductor cable.

MULTIPLAYER GAMES

With the ACCESS.bus hardware interface defined and stable, software application opportunities are now emerging. Three key market areas have initial momentum in application development: multiplayer PC games, training and simulation, and education.

Computer Access Technology Corporation (CATC) developed two multiplayer games to show the features and benefits of the ACCESS.bus. The Windows versions of blackjack and Chinese checkers host up to six simultaneous players. These games highlight the simultaneous and independent interaction of multiple players, each with his or her own I/O devices. Both these games use multiple mice as their primary input device. Joysticks, trackballs, tablets, or arrow keys can also serve as the device of choice.

These games demonstrate ACCESS.bus technology, offering insight into additional game possibilities. Interactive video games, in which combatants participate seated adjacent to each other and not through a network link, offer participants the opportunity to heighten reactive and cognitive-interactive computer skills while also experiencing the emotional intensity of shoulder-to-shoulder competition and collaboration.

EDUCATIONAL SOFTWARE

Though most people queried believe that computer games are more recreational than educational, early indications of the educational benefits of games using interactive computer skills show that games enhance learning.

ABIG hosted a Software Creators Contest over Internet. Several abstracts for ACCESS.bus software applications were submitted for review. ACCESS.bus technology games were predominant. Of the submissions, four finalists were picked. Finalists were chosen according to the perceived functionality and benefit of their ideas. Notably, the proposals' underlying theme involved interactive learning tools disguised as games.

Imagine the benefits of having the entire classroom connected to a single CPU. Schools would only need to provide individual keyboards and displays, not an entire CPU, for each student. Interactive teaching, on-line tutorials, and instant feedback on lesson plans are only some of the potential benefits. It would be easier to monitor the students' learning process. Student-specific course curricula could be administered and monitored with greater expediency and accuracy. Most importantly, the budget required to supply a single school with computers could be spread across the entire school district.

With ACCESS.bus, educational applications for text and document storage and retrieval now offer a viable alternative to the hard-bound book. Standardized achievement tests can be distributed, graded, and the results tabulated with greater simplicity and less cost. While the technologist and entrepreneur might find this new market opportunity appealing, so does the taxpayer who endorses the applications because it makes better use of the public coffers and increases the potential educational benefit and knowledge retention of the student.

TRAINING AND SIMULATION SOFTWARE

ACCESS.bus is also useful for more advanced forms of training and technical orientation. The first flight simulator available to the desktop computer was a derivative of an actual military training tool. Unfortunately for the simulation program, the "student" was only able to play against the computer and not other humans. No amount of "fuzzy logic" is yet able to adequately imitate the dynamic of human behavior. Virtual reality does offer significant promise in allowing many people to interact with each other in simulated training or game conditions.

ACCESS.bus offers the virtual developer a complete hardware, software, and application layer interface with an easy and cost-effective cabling and connector interface. The programmer can realize the full potential of a given creation without prohibitive implementation costs.

ACCESS.bus LAYERING

The ACCESS.bus standard includes a physical layer as well as several software layers. The physical layer defines the transmission medium and connectors for the bus (electric signals, cables, and connectors) and the basic message format (Start, Stop, Acknowledge, Arbitration, etc.). In addition, ACCESS.bus specifications describe base and application protocols in communications between peripheral devices and the code on a host computer.

THE PHYSICAL LAYER

At the hardware level, ACCESS.bus physical layer is based on the well-established Inter-Integrated Circuit (I^2C) serial bus developed by Philips/Signetics. The serial bus architecture features a single data line which

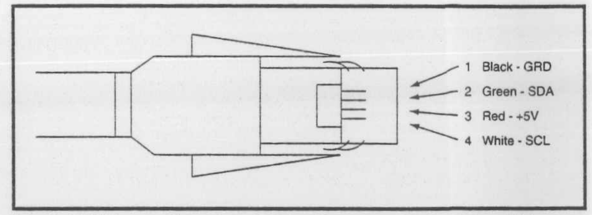

Figure 2—ACCESS.bus uses a four-conductor, shielded, modular connector similar to a telephone RJ-11.

Byte Number	7 (MSB)	6	5	4	3	2	1	0 (LSB)	
1	destaddr							0	Destination address
2	srcaddr							0	Source address
3	P	length							Protocol flag, message length
4	body								1 to 127 bytes
⋮	⋮								
Length + 4	checksum								

Figure 3—*The ACCESS.bus message packet includes a standard header, a variable-length body, and a checksum.*

carries one bit of information at a time. This, of course, results in lower costs for cabling, connectors, and controller circuitry.

The simple and efficient I²C protocol defines a symmetric, multi-master bus on which arbitration among contending masters is effected without losing data. I²C cooperatively synchronizes the serial clock for exchange of data between bus partners with different maximum clock rates. Addressing, framing of bits into bytes, and byte-acknowledgment by the receiver are all defined by a bus-transaction scheme within the I²C protocol. I²C microcontrollers handle the logical requirements of bit-level handshaking.

Pictured in Figure 2 is the ACCESS.bus physical connection. The shielded cable contains four wires: serial data (SDA), serial clock (SCL), power (+5 V), and ground (GND). It uses standard, shielded, modular connectors available from AMP and Molex. This shielding of the cables and connectors enables ACCESS.bus to meet FCC radiation and ESD requirements.

A typical ACCESS.bus device has two connectors so that devices may be chained on the single bus. Hand-held devices may have a captive cable joined to the bus trunk with a T connector. The serial-data (SDA) and serial-clock (SCL) lines work together to define the information carried on the bus.

The physical layer can support clock rates up to 100 kbps.

BASE PROTOCOL

The base protocol establishes the nature of ACCESS.bus as an asymmetrical interconnect between a host computer and a number of peripheral devices. The host plays a special role as a manager of the ACCESS.bus. Data communication is always between host and peripheral device and never between two peripherals. Although the I²C protocol provides for mastery by either the sender or the receiver of a bus transaction, the ACCESS.bus protocol defines masters as exclusively senders and slaves as exclusively receivers. Of course, the host and all the devices are both master senders and slave receivers at different times.

The ACCESS.bus base protocol defines the format of an ACCESS.bus message envelope, which is an I²C bus transaction with additional semantics, including checksum reliability control. The base protocol defines a set of control and status message types, which are used in the configuration process.

Figure 3 gives the ACCESS.bus message format. The first byte in the message is the receiver's unique address. The second byte contains the transmitter's unique address. The third byte of a message comprises two fields. Bits 2–7 provide a byte count for the body of the message. Thus, a message body can have 0–127 bytes and is followed by a checksum byte for error control. The checksum is the bitwise `XOR` of all the preceding bytes of the message.

The high-order bit of the third byte is a protocol flag (P), which distinguishes between data stream messages (P = 0) and control and status messages (P = 1). Datastream messages carry the application information exchanged between the device and the host. The control and status messages manage the ACCESS.bus protocol. As well, the base protocol defines a set of nine control and status message types used in the configuration process (see Table 1).

Two of the unique features of this configuration process include *autoaddressing* and *hot plugging*. With autoaddressing, devices are assigned unique bus addresses in the configuration process without the need for setting jumpers or switches on the devices. Hot plugging enables devices to be attached or detached while the system is running.

APPLICATION PROTOCOL

The application protocol is the highest level of the ACCESS.bus protocol. It defines message semantics that are specific to particular functional types of devices. Different device types require different application protocols.

So far, application protocols have been defined for three device classes: keyboards, locators, and text devices. Each of these predefined classes is designed to be broad. The keyboard device protocol, for instance, defines standard messages for reporting keystrokes and controlling keyboard peripherals. The protocol attempts to

a)

Computer-to-device Messages	Message Purpose
`Reset()`	Force device to power-up state and default ACCESS.bus address.
`Identification Request()`	Ask device for its identification string.
`Assign Address(ID strng, new addr)`	Tell device with matching identification string to change its address to new address.
`Capabilities Request(offset)`	Ask device to send the fragment of its capabilities information that starts at offset.
`Enable Application Report`	Enable or disable a device to send application reports to the host computer.
`Presence Check`	Check if the device is present on the bus at the specific address.

b)

Device-to-computer Messages	Message Purpose
`Attention`	Inform computer that a device has finished its powerup or reset test and needs to be configured.
`Identification Reply(ID string)`	Reply to identification request with device's unique identification string.
`Capabilities Reply(offset, data frag)`	Reply to capabilities request with data fragment which includes a fragment of the device's capabilities string. The computer uses offset to reassemble fragments.

Table 1—*The ACCESS.bus base procotol defines nine control and status messages used in the configuration process.*

define the simplest set of functions from which industry-standard keyboard interfaces can be built.

The locator device protocol defines a set of standard messages for reporting locator movement and keyswitch activation for mice, tablets, and other positioning devices. Complex locator devices can be modeled as a combination of basic devices or as their own device driver.

The text device protocol provides a simple way to transmit character or binary data to and from stream-oriented devices such as a printer, bar-code reader, or modem. The sequential-character-stream model also serves as a common denominator for connecting RS-232 interface devices.

Designing devices that conform to the general device semantics is a major advantage. Device-specific software, both in the device-resident firmware and the driver software needed for the host operating system, can be accessed by the new device.

In the future, it is anticipated that more device-specific application protocols will be defined under the aegis of the ABIG. As well, any device vendor may implement a special device protocol within the general message envelope defined by the base protocol.

Participation in all three of the protocol levels requires understanding of the device level. The lower levels of this firmware are likely to be common to many devices. Higher levels of the firmware are expected to be more specific to the device and the application.

SOFTWARE ARCHITECTURE

Figure 4 depicts the host software structure of the ACCESS.bus. An ACCESS.bus peripheral requires software at both ends of the bus transaction for managing all levels of the peripheral interaction—the physical layer, base protocol, and application protocol. As well, the peripheral device requires software to support communication between the device microcontroller and the application-specific I/O transducer circuitry.

The host-system operating software must provide interfaces so that application programs can access both ACCESS.bus devices and the ACCESS.bus itself. Since the lower levels of the interaction are common to diverse device types, they can be supported by the same or similar software modules.

The Mini Port Driver (MPD), pictured in Figure 5, is a communication software layer that separates the ACCESS.bus specific hardware from the ACCESS.bus manager.

The manager is a central software driver that controls the operation of all ACCESS.bus devices attached to the bus. It communicates with the Mini Port Driver on the one side and with the ACCESS.bus device drivers on the other.

The manager initializes and controls the ACCESS.bus, recognizing newly inserted or removed devices. It links device drivers and applications with specific ACCESS.bus devices, validates incoming messages, and serves as a bidirectional data switch routing messages to and from the appropriate device(s).

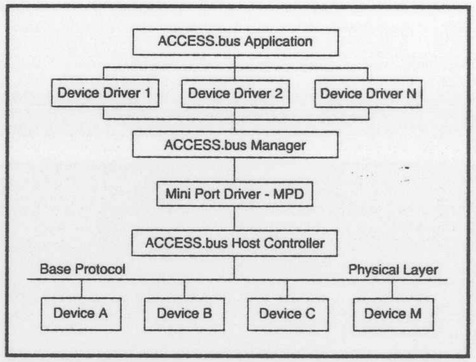

Figure 4—*The application software running on the host is shielded from the ACCESS.bus controller by several layers of software.*

SOFTWARE DEVICE DRIVERS

The software device drivers serve as a bidirectional interface between application programs and a specific type of device or devices (mouse driver, keyboard driver, etc.). Obviously, the appropriate driver depends on the device type. However, there may be further parameters that characterize the device and affect the choice of driver or specific arguments for a selected driver.

It is important to remember that the application program may also need to be informed of these device parameters. The device-capabilities-information feature of the ACCESS.bus protocol gives a measure of device independence in the selection of drivers and informs the host software of device characteristics.

APPLICATION LAYER

All application programs communicate with ACCESS.bus devices via an ACCESS.bus device driver or directly with the bus manager. Applications can use many devices of the same type (multiple mice, multiple keyboards).

TRANSFERRING DATA TO AND FROM APPLICATIONS

For a device to be accessible to application programs running on the host, it must be connected to an appropriate software driver. Establishing this association is the last phase of configuration.

Device-capabilities information explicitly states a device's functional characteristics. Although these characteristics are implicit in the device-type designation contained in the ID string, there are sometimes variances among individual devices of a given type. For example, the capabilities information might include the national alphabet for a keyboard or resolution and units for a locator.

The capabilities information is contained in an ASCII-encoded text string stored on the device ROM. The base protocol defines a simple and compact grammar for building the capabilities string.

The semantics are carried by keywords. The base protocol defines

keywords which apply to all device types. The application protocol defines the keywords specific to certain device types. To date, the application protocols define semantics for the capabilities information for generic keyboards, locators, and general text devices. The grammar allows for easy extension of the capabilities-information specification.

THE MOUSE DRIVER

The ACCESS.bus Multiple Mouse Driver (ABMSWIN.DLL) is a driver developed by CATC and offered as part of their Windows application development kit.

The driver connects to as many devices as specified in the [AbMsWin] section under the maxDev entry. The application connects to this driver by supplying the target queue (HWND) for mice movements and button state. Interrupts are converted to Windows messages and placed in the application-input queue. These messages are in raw format and cannot be used by the application.

To convert the raw message into a more useful format, the application returns the messages to the driver, which in turn processes the information and sends additional messages to the application message queue. The messages include information such as moves, button state, connect, disconnect, and so on. The processed messages are in the following format:

wParam:
LOBYTE is the mouse ID
HIBYTE is the button state
lParam:
LOWORD is the x position in pixels
HIWORD is the y position in pixels

There are certain restrictions that the programmer must plan for:

- The driver blocks information from the application once a raw message is sent. To continue to receive information, the application must return the raw message to the driver. This feature prevents overflow of the message queue.
- The driver is capable of handling one window at a time. Mouse bitmaps are displayed in the client area of the connected window.
- Application information is saved and restored when the mouse is in motion, but there is no protection to the mouse image. The application must explicitly turn mice off prior to screen update, and then turn them on again.

MOUSE DRIVER SERVICES

Upon interrupt, the application is notified with a raw message, "type MICE_EVENT" which is defined as WM_USER+128. The application must then send this information to the driver using the AbMouseMoveMouse service. The driver processes the raw information and finally replies with one of the messages listed in Table 2.

Specific service messages are also necessary for complete application compatibility. Many of the service messages and their meanings are included in Table 3.

THE KEYBOARD DRIVER

The ACCESS.bus Multiple-Keyboard Driver (ABKBWIN.DLL), which is included in CATC's application development kit, connects to as many devices as specified in the maxDev entry of the [AbKbWin] section. The application connects to this driver by supplying the target queue (HWND) for keyboard events. Interrupts and the resulting messages are processed in a manner similar to that described for mouse drivers..

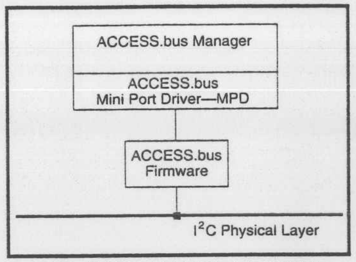

Figure 5—*The ACCESS.bus Mini Port Driver is the ACCESS.bus firmware-specific interface. The bus manager layer remains constant despite host firmware adaptions.*

CHINESE CHECKERS

Chinese checkers is a native Windows game application. Similar to the board game, the application can be played by two to six players. Ten marbles are "placed" in the player's section of a six-pointed star. Each group of marbles is a color unique from the rest of the groups and the player of a group is represented by a same-color pointing device (i.e., only the player with the yellow cursor can move the yellow marbles). The number of players can be changed, and one or more groups can be assigned to the computer using the Game Setup command. (Previous to the ACCESS .bus implementation, Chinese checkers was a single-cursor game in which one cursor could control and move any and all marbles.)

The setup window is simple and self explanatory. A group of marbles can be added or removed, assigned to an individual or the computer, allotted a turn sequence, and changed color. The addition or deletion of a mouse (or another locating device) automatically adds or deletes a cursor and group of marbles for the associated player.

Each one of the active mice moves a colored cursor while it is inside the game window. You can turn the colored cursor into a Windows system cursor by moving it out of the game window from the top side of the window (the menu side). Once the cursor is out of the game window, it becomes a standard Windows mouse cursor. When you move it back into the application window, it automatically reverts to a colored game cursor.

BLACKJACK

The ACCESS.bus version of blackjack was created specifically to show the simultaneous, multiplayer capacity of the ACCESS.bus. The game window gives a top-down view of a blackjack table in a casino with stations for six players, each having their own betting chips and cursors.

Unlike Chinese checkers, which operates in serial mode, each player playing in turn, blackjack is a parallel game—participants play simultaneously. Each participant can concurrently place a wager, buy more chips from the cashier, and indicate a "hit" or "stand" from the dealer.

All setup and operational characteristics associated with the Chinese checkers apply to blackjack. The game includes on-line help, configuration of mice and colors, rules of the game, changing active mouse buttons, and alteration of card design.

DESIGNING A GAME

Not only is the Chinese checkers an adaptation of a classic board game, it is an example of converting an existing game into an ACCESS. bus game. When David Williams originally created the Chinese checkers computer game, he intended it to be a single-input-device game. Whether play was against the computer or other humans, only one interface device could be used by participants. As one move completed, the next person took over the mouse and made his or her move.

The opportunity for each participant to "own" his or her pointing device (mouse, trackball, or joystick) and the ability of the software to accept and accommodate actions of several devices brings a whole new look and feel to the game. No longer do participants need to clamor for their turn at the mouse. Players can even use the pointing device of their choice.

MICROCODE DEVELOPMENT

The microcontroller in the device provides the intelligence for managing the device's participation in all levels of the ACCESS.bus protocol. Use of components with I^2C interface functionality simplifies development of the lowest level of the interface and protocol. Since the protocol concerns only the bus communication methods common to all sorts of peripheral devices, they may be implemented by reusing software previously developed for some of these components. Devices conforming to the semantics of the predefined standard application protocols may also benefit from the availability of some off-the-shelf code at the top level.

```
MICE_LDOWN        (MICE_EVENT+1)    /* left button pressed */
MICE_LDBLCLK      (MICE_EVENT+2)    /* left button dblclk */
MICE_LUP          (MICE_EVENT+3)    /* left button released */
MICE_MDOWN        (MICE_EVENT+4)    /* middle button pressed */
MICE_MDBLCLK      (MICE_EVENT+5)    /* middle button dblclk */
MICE_MUP          (MICE_EVENT+6)    /* middle button releases */
MICE_RDOWN        (MICE_EVENT+7)    /* right button pressed */
MICE_RDBLCLK      (MICE_EVENT+8)    /* right button dblclk */
MICE_RUP          (MICE_EVENT+9)    /* right button released */
MICE_MOVE         (MICE_EVENT+10)   /* mouse move */
MICE_CONNECT      (MICE_EVENT+11)   /* mouse connected */
MICE_DISCONNECT   (MICE_EVENT+12)   /* mouse disconnected */
```

Table 2—*Upon a mouse interrupt, the application is notified of a mouse event with a raw message. Once that message is passed to the driver, the driver responds with one of these messages.*

`void far pascal AbMouseDriverOn(HWND hwndApp, HDC hdc)`
Invoked at initialization to indicate the target window for mice events. Zero is returned for success, and a negative number if the driver is already busy with another application.

`void far pascal AbMouseDriverOff(void)`
Disconnects the window from the driver.

`int far pascal AbMouseActiveDevs(void)`
Returns the number of connected devices.

`int far pascal AbMouseGetPos(int id, int far *btn, int far *x, int far *y)`
Returns the mouse position and button state. If the mouse is disconnected, then the service returns zero. If the mouse is connected, then the service returns a nonzero value. *btn*, *x*, and *y* are set with the button state and the mouse position. Buttons are represented by bits: D0 = left button, D1 = right button, and D2 = middle button.

`void far pascal AbMouseXorMouse(HDC hdc, int id)`
XORs the mouse bitmap at the current position.

`void far pascal AbMouseMoveMouse(HDC hdc, WPARAM wParam, LPARAM event)`
Must be called for each raw message from the driver. The wParam and lParam parameters should be identical to the values passed via the `MICE_EVENT` message.

`void far pascal AbMouseClientArea(RECT far *rect)`
Informs the driver of the client area size. The mice images are restricted to the client area of the connected window. Invoke this service upon the `WM_SIZE` message.

`void far pascal AbMouseSetPos(int id, int x, int y)`
Sets a particular mouse position. At initialization, the position is set to x = 0 and y = 0. The application may use this service to center the mouse in the client area. If a device was connected while the application was running, the driver assigns the position as the center of the client area.

`int far pascal AbMouseGetSysMouse(void)`
Instructs the system mouse driver (`ABMOUSE.DLL`) to release one of its devices and it releases the last active device. The multiple device driver (`ABMSWIN.DLL`) then attempts to claim that mouse. Applications can use this routine to convert a system mouse into an application mouse. The returned value is negative if no slots are available in `ABMSWIN.DLL` driver. In this case, you can increase the number of the devices in `maxDev` of the `[AbMsWin]` section of your `ACCBUS.INI`. However, the normal return is zero which the service returns even if no mouse is found (this occurs when no system mouse is available). An application may test for this condition using the `AbMouseActiveDevs()` services.

`int far pascal AbMouseRelSysMouse(int id)`
Causes `ABMSWIN.DLL` to release a device and `ABMOUSE.DLL` to claim it. Applications may use this service to convert application mice to system mice. Zero is returned for success, and a negative value is returned if the device identification is invalid or not connected.

`void far pascal AbMouseSetBitmap(HDC hdc, int id, HBITMAP hBmp)`
Sets the mouse image. The mouse image is a 16 ¥ 16 bitmap that was loaded using Windows' `LoadBitmap()`. Note that the driver will not delete the bitmap upon termination. It is the responsibility of the application to call Windows' `DeleteObject()`. The returned value is 0 if the function is successful and –1 if the device identification is invalid.

`int far pascal AbMouseRestoreBitmap(HDC hdc, int id)`
Restores the default mouse image. Zero is returned for success, and –1 for an invalid device identification.

`int far pascal AbMouseShortCapabilities(int id, DevProt far *devProt)`
Fills the given structure devProt with the device's prot, type, and model strings (see ACCESS.bus specification for the meaning of these parameters). Zero is returned for success and –1, for invalid device identification. This service may be invoked to determine the specific model of a device

Table 3—*Under the ACCESS.bus protocol, service messages are defined to handle all aspects of the operation of attached devices. This is just a sampling of some of the messages available to do mouse support.*

HOST SOFTWARE DEVELOPMENT

Vendors of host systems supporting ACCESS.bus must supply drivers and other operating systems modules necessary for access to the ACCESS.bus port both for application program clients and for other system software such as the interactive I/O handlers of the window system. Here again, many ABIG member companies offer a wide variety of products to support ACCESS.bus functionality, including a manager, mini port and device drivers for various operating systems, software development kits, and source-code modules.

OTHER GAMES

What fun is a game that cannot be played with other people?

Chinese checkers and blackjack are small examples of the feature characteristics that ACCESS.bus offers the game and entertainment industry. Skill games especially, like chess or checkers, can be played by humans in front of a single video display without the need to either share an input device or be relegated to competition with a computer.

Other classic casino games—poker, roulette, and craps—are natural candidates for ACCESS.bus support as these games typically have several participants involved at one time. As in blackjack, each player needs individual control over the amounts and placement of their betting chips. Simultaneous placement of wagers, cashier interaction, and execution of play would be easy and effective in the ACCESS.bus environment.

Chinese checkers offers insight into other board games such as Monopoly, Life, or Clue. Although some of these are available for computer, they don't offer player interactivity. The games would gain new life with control devices for each player. Players would gain more control over piece movement and game interaction.

The largest game-growth sector, however, is the action-game market. Imagine SuperMario with two or more plumbers or a multiplayer Wolfenstein. The interactive opportunity of these types of games would open new markets to the manufacturers of both the games and peripherals.

With ACCESS.bus, the software company would no longer be confined to building games with the idea that only one mouse, joystick, keyboard, or trackball can be attached to the system. In fact, the game company could bundle new peripherals with the game at attractive prices and be assured that the new devices would be compatible with the system.

BEYOND GAMES

ACCESS.bus is an evolution of peripheral connection technology for the personal computer industry. The ability to add or delete devices from a system is only the first step. Intelligent components of the entire systems may soon have plug-and-play support characteristics. The ability to plug in a memory module or coprocessor, turn on the system, and have the new components recognized with the power-on self test is a near reality.

The Video Exchange Standards Association (VESA) has adopted ACCESS.bus for use in display devices. The display data channel (DDC) specification incorporates ACCESS.bus as a control interface for autoconfiguration of a compliant display device to a host computer system. The DDC also facilitates control of the display adjustments—refresh rate, vertical and horizontal scan, resolutions, color temperatures, screen position, brightness, contrast, and other characteristics—and lets users create software files that load user-defined or preset configurations to the monitor. Video adapter vendors may offer ACCESS.bus host interfaces in their products or pass the ACCESS.bus information to another device such as an add-on card, adapter, or system motherboard.

Eventually, it may be possible to hide the computer under or behind the desk. Peripherals will be connected to an ACCESS.bus port on the front of the monitor. The need to run several cables out the back of the system onto the desk will be eliminated.

OTHER APPLICATIONS

While games are attractive and splashy, many other applications appear to benefit from the use of ACCESS.bus. At this time, the most pressing need is educational software.

In the classroom, where computer literacy is as much a part of the course curriculum as math and language, the opportunity to place a computer system at every student's desk is not feasible due to cost, configuration, and connectivity. Even in today's market of low-cost, preconfigured, off-the-shelf systems, the typical Windows-compatible personal computer is about $1,000. If you multiply that by 500–1000 students per school, the overall cost could approach $1,000,000 per school! Add in the cost of maintenance and support—the computing classroom becomes untenable.

Conversely, educational software programs that serve an entire class from one computer system makes the possibility of the computing classroom much more feasible. A keyboard with a small LCD panel and an appropriate software application would let the teacher engage the class in a host of on-line activities.

Computer systems manufacturers are developing these hardware solutions today. What's needed the most is a software base!

THE FUTURE OF ACCESS.BUS

ACCESS.bus is already a proven, stable technology with a strong base in embedded control and vertical applications. Even without the opportunities of the game and educational markets, the possibility of attaching multiple peripherals to the power of the latest wave of high-speed CPUs is attractive. The system board and CPU certainly have enough bandwidth and power to support multiple concurrent tasks.

Why limit multitasking to four COM ports and a couple of parallel ports? ACCESS.bus not only provides system connectivity, but may in fact obviate COM ports and conventional peripheral connectivity arrangements.

The fact that ACCESS.bus is an open industry standard providing an easy and effective connection to multiple devices ensures its use and growth in scope and appeal.

In fact, now is a good time to initiate a ride on the ACCESS.bus wave since the field is not crowded. The hardware cost benefits are such that new software and applications will arguably achieve greater profit margins and quicker market acceptance than new nonACCESS.bus applications.

David Rodgers is the director of sales for Computer Access Technology. After starting his career as an engineering technician in 1983, he has worked as a national sales and marketing director for several large companies.

Peretz Tzarnotzky is vice-president of engineering at Computer Access Technology and has 22 years experience as an electronics and systems engineer.

Either author can be reached at catc@netcom.com.

CONTACTS

For full details about the Software Creators Contest:

ACCESS.bus Industry Group
370 Altair Way, Ste. 215
Sunnyvale, CA 94086
(408) 991-3517
Fax: (408) 991-3773
ABIG@netcom.com

For complete systems solutions including development tools for hardware, software, and firmware.

Computer Access Technology Corp.
3375 Scott Blvd., Ste. 410
Santa Clara, CA 95054
(408) 727-6600
Fax: (408) 727-6622
catc@netcom.com

For educational software:

Emerald City Education
Lowry Building, Ste. 103
700 North Egan Rd.
Madison, SD 57042
(605) 256-5681

I R S

404 Very Useful
405 Moderately Useful
406 Not Useful